

# Other Stuff for Overture

## User Guide, Version 1.0

William D. Henshaw <sup>1</sup>

Centre for Applied Scientific Computing  
Lawrence Livermore National Laboratory  
Livermore, CA, 94551  
henshaw@llnl.gov  
<http://www.llnl.gov/casc/people/henshaw>  
<http://www.llnl.gov/casc/Overture>

June 10, 2002

UCRL-MA-134292

**Abstract:** We describe miscellaneous Overture stuff:

**Overture start and finish functions, Overture global variables**

**sPrintF,sScanF,...** : miscellaneous utility routines.

**getIndex** : the getIndex utility routines for generating A++ Index's for operating on sub-domains of MappedGrid's.

**Twilight-zone functions** : OGFunction, OGPolyFunction, OGTrigFunction and OGPulseFunction.

**Database access functions** : functions for reading various objects (such as a CompositeGrid) from a database file, including the function getFromADataBase.

**display functions** : formatted display functions for A++ arrays and for writing A++ arrays to files.

**Integrate** : This class provides methods for easily integrating grid functions on domains or boundaries of domains.

**TridiagonalSystem solver** : solve tridiagonal and block tridiagonal systems.

**FortranIO** : write unformatted Fortran files.

**Reference Counting** : A description of how reference counting works.

---

<sup>1</sup>This work was partially supported by grant N00014-95-F-0067 from the Office of Naval Research

# Contents

<b>1 Overture start and finish functions, Overture global variables</b>	<b>5</b>
1.1 Overture global variables . . . . .	5
1.2 start . . . . .	5
1.3 finish . . . . .	5
1.4 getGraphicsInterface . . . . .	6
1.5 setGraphicsInterface . . . . .	6
1.6 getMappingList . . . . .	6
1.7 setMappingList . . . . .	6
1.8 setDefaultGraphicsParameters . . . . .	6
<b>2 Miscellaneous Stuff</b>	<b>7</b>
2.1 sPrintF . . . . .	7
2.2 sScanF . . . . .	7
2.3 sScanF . . . . .	8
2.4 getLine . . . . .	8
2.5 getLine . . . . .	8
2.6 ftor . . . . .	9
<b>3 Constructing A++ Index Objects for Grid Functions: getIndex, getBoundaryIndex, getGhostIndex</b>	<b>10</b>
3.1 Index functions . . . . .	10
3.2 getIndex from an index array . . . . .	10
3.3 getIndex from a {float,double,int}MappedGridFunction . . . . .	11
3.4 getBoundaryIndex from an index array . . . . .	11
3.5 getBoundaryIndex from a {float,double,int}MappedGridFunction . . . . .	12
3.6 getBoundaryIndex from a {float,double,int}MappedGridFunction . . . . .	12
3.7 getGhostIndex from an index array . . . . .	13
3.8 getGhostIndex from a {float,double,int}MappedGridFunction . . . . .	13
3.9 getIndex from a {float,double,int}MappedGridFunction . . . . .	15
3.10 getBoundaryIndex from a {float,double,int}MappedGridFunction . . . . .	15
3.11 getBoundaryIndex from a {float,double,int}MappedGridFunction . . . . .	16
3.12 getGhostIndex from a {float,double,int}MappedGridFunction . . . . .	16
3.13 getIndex from a {float,double,int}MappedGridFunction . . . . .	18
3.14 getBoundaryIndex from a {float,double,int}MappedGridFunction . . . . .	18
3.15 getBoundaryIndex from a {float,double,int}MappedGridFunction . . . . .	19
3.16 getGhostIndex from a {float,double,int}MappedGridFunction . . . . .	19
3.17 getIndex from a {float,double,int}MappedGridFunction . . . . .	21
3.18 getBoundaryIndex from a {float,double,int}MappedGridFunction . . . . .	21
3.19 getBoundaryIndex from a {float,double,int}MappedGridFunction . . . . .	22
3.20 getGhostIndex from a {float,double,int}MappedGridFunction . . . . .	22
<b>4 OGFunction: A class for defining Twilight-zone flows</b>	<b>24</b>
4.1 Evaluate the function or a derivative at a point . . . . .	24
4.2 Evaluate the function or a derivative on a MappedGrid . . . . .	25
4.3 Evaluate the function or a derivative on a CompositeGrid . . . . .	26
4.4 OGPolyFunction . . . . .	28
4.4.1 Constructor . . . . .	28
4.4.2 setCoefficients . . . . .	28
4.5 OGTrigFunction . . . . .	29
4.5.1 Constructors . . . . .	29
4.5.2 setAmplitudes . . . . .	29
4.5.3 setConstants . . . . .	29
4.5.4 setFrequencies . . . . .	29
4.5.5 setShifts . . . . .	30
4.6 OGpulseFunction . . . . .	31
4.6.1 Constructor . . . . .	31
4.6.2 setRadius . . . . .	32

4.6.3	setRadius . . . . .	32
4.6.4	setCentre . . . . .	32
4.6.5	setVelocity . . . . .	32
4.6.6	setCoefficients . . . . .	32
4.7	Examples . . . . .	33
<b>5</b>	<b>Data-base Access Functions</b>	<b>34</b>
5.1	getFromADataBase(CompositeGrid & cg,...) . . . . .	34
<b>6</b>	<b>Display functions for arrays (writing arrays to files)</b>	<b>35</b>
6.1	display: display an A++ array . . . . .	35
6.2	display: save an A++ array in a file . . . . .	35
6.3	display an A++ array with DisplayParameters . . . . .	35
6.4	displayMask . . . . .	35
<b>7</b>	<b>Integrate: integrate grid functions on overlapping grids</b>	<b>36</b>
7.1	Member Functions . . . . .	36
7.2	constructor . . . . .	36
7.3	constructor . . . . .	36
7.4	updateToMatchGrid . . . . .	36
7.5	defineSurface . . . . .	36
7.6	numberOfFacesOnASurface . . . . .	37
7.7	getBoundaryDefinition . . . . .	37
7.8	getFace . . . . .	37
7.9	integrationWeights . . . . .	37
7.10	leftNullVector . . . . .	37
7.11	surfaceIndex . . . . .	37
7.12	surfaceIntegral . . . . .	38
7.13	surfaceIntegral . . . . .	38
7.14	volumeIntegral . . . . .	38
<b>8</b>	<b>TridiagonalSolver: Solve sets of tridiagonal systems or block tridiagonal systems</b>	<b>39</b>
8.1	Member Functions . . . . .	39
8.2	constructor . . . . .	39
8.3	factor . . . . .	40
8.4	solve . . . . .	41
8.5	solve . . . . .	41
<b>9</b>	<b>FortranIO : Write Fortran formatted or unformatted files from C++</b>	<b>42</b>
9.1	constructor . . . . .	42
9.2	open . . . . .	42
9.3	close . . . . .	42
9.4	print( int ) . . . . .	42
9.5	print( float ) . . . . .	42
9.6	print( double ) . . . . .	42
9.7	print( int* ) . . . . .	43
9.8	print( float* ) . . . . .	43
9.9	print( double* ) . . . . .	43
9.10	print( aString ) . . . . .	43
9.11	print( intArray ) . . . . .	43
9.12	print( floatArray ) . . . . .	43
9.13	print( doubleArray ) . . . . .	43
9.14	print( intArray,floatArray ) . . . . .	43
9.15	print( intArray,doubleArray ) . . . . .	43
9.16	read( int ) . . . . .	44
9.17	read( float ) . . . . .	44
9.18	read( double ) . . . . .	44
9.19	read( int* ) . . . . .	44

9.20 read( float* ) . . . . .	44
9.21 read( double* ) . . . . .	44
9.22 read( aString ) . . . . .	44
9.23 read( intArray ) . . . . .	44
9.24 read( floatArray ) . . . . .	44
9.25 read( doubleArray ) . . . . .	45

<b>10 Reference Counted Objects</b>	<b>46</b>
10.1 Introduction . . . . .	46
10.2 How to Write a Reference Counted Class . . . . .	46
10.3 Class ListOfReferenceCountedObjects . . . . .	49
10.3.1 Constructors . . . . .	49
10.3.2 Public Member Functions . . . . .	49
10.3.3 Examples . . . . .	50

# 1 Overture start and fi nish functions, Overture global variables

## 1.1 Overture global variables.

The Overture class contains global variables that can be used as default arguments to functions. For example, `Overture::nullRealArray`, can be used as a default argument for a `RealArray`. These instances of classes are accessed as function calls as opposed to building static global variables. Initially the later approach was taken but this caused difficulties since the loader would build the classes in some unknown order.

```
floatSerialArray & nullFloatArray():

doubleSerialArray & nullDoubleArray():

RealArray & nullRealArray():

intSerialArray & nullIntArray():

floatDistributedArray & nullFloatDistributedArray():

doubleDistributedArray & nullDoubleDistributedArray():

RealDistributedArray & nullRealDistributedArray():

IntegerDistributedArray & nullIntegerDistributedArray():

MappingParameters & nullMappingParameters():

floatMappedGridFunction & nullFloatMappedGridFunction():

doubleMappedGridFunction & nullDoubleMappedGridFunction():

realMappedGridFunction & nullRealMappedGridFunction():

intMappedGridFunction & nullIntMappedGridFunction():

floatGridCollectionFunction & nullFloatGridCollectionFunction():

realGridCollectionFunction & nullRealGridCollectionFunction():

doubleGridCollectionFunction & nullDoubleGridCollectionFunction():

intGridCollectionFunction & nullIntGridCollectionFunction():

BoundaryConditionParameters & defaultBoundaryConditionParameters():

GraphicsParameters & defaultGraphicsParameters():
```

## 1.2 start

```
int
start(int argc, char *argv[])
```

**Description:** Overture initialization function. Call this routine before calling any Overture functions.

## 1.3 fi nish

```
int
finish()
```

**Description:** Overture cleanup function. Call this routine when you are done using Overture.

## **1.4 getGraphicsInterface**

```
GenericGraphicsInterface*
getGraphicsInterface(const aString & windowTitle/*="Your Slogan Here"*/, const bool initialize/*=true*/,
                     int argc/* = 0*/, char *argv[] /*=NULL*/)
```

**Description:** Return a pointer to the one and only graphics interface. If the pointer is null, a new graphics interface will be built.

## **1.5 setGraphicsInterface**

```
void
setGraphicsInterface( GenericGraphicsInterface *ps)
```

**Description:** Set the default graphics interface. This is normally called by the GenericGraphicsInterface constructor, there is no need for a typical user to call this function.

## **1.6 getMappingList**

```
ListOfMappingRC*
getMappingList()
```

**Description:** Return a pointer to the default list of mappings. This pointer may be NULL.

## **1.7 setMappingList**

```
void
setMappingList(ListOfMappingRC *list)
```

**Description:** Set the default list of mappings.

## **1.8 setDefaultGraphicsParameters**

```
void
setDefaultGraphicsParameters( GraphicsParameters *gp =NULL)
```

**Description:** Set the default graphics parameters. By default reset the graphics parameters to the standard one.

## 2 Miscellaneous Stuff

### 2.1 sPrintF

aString &  
sPrintF(aString & s, const char \*format, ...)

**Description:** Implementation of an "sprintf" like function that returns the formatted string s and NOT the number of chars assigned.

**NOTE:** this function assumes a maximum of 300 chars in the format string.

**s (input) :** fill in this string.

**format (input) :** use this printf style format.

**argument '...' (input):** variable length argument list.

**author:** wdh

### 2.2 sScanF

```
int  
sScanF(const aString & s, const char *format,  
       void *p0,  
       void *p1=NULL,  
       void *p2=NULL,  
       void *p3=NULL,  
       void *p4=NULL,  
       void *p5=NULL,  
       void *p6=NULL,  
       void *p7=NULL,  
       void *p8=NULL,  
       void *p9=NULL,  
       void *p10=NULL,  
       void *p11=NULL,  
       void *p12=NULL,  
       void *p13=NULL,  
       void *p14=NULL,  
       void *p15=NULL,  
       void *p16=NULL,  
       void *p17=NULL,  
       void *p18=NULL,  
       void *p19=NULL,  
       void *p20=NULL,  
       void *p21=NULL,  
       void *p22=NULL,  
       void *p23=NULL,  
       void *p24=NULL,  
       void *p25=NULL,  
       void *p26=NULL,  
       void *p27=NULL,  
       void *p28=NULL,  
       void *p29=NULL  
)
```

**Description:** A special version of sscanf that strips out any ',' characters and replaces them with ' ' and converts the format string with ftoi so that %e and %f formats are converted properly for double precision.

**p0,p1,... (input) :** supply addresses of variables to save the results in.

### 2.3 sScanF

```
int  
fScanF(FILE *file, const char *format,  
        void *p0,  
        void *p1 =NULL,  
        void *p2 =NULL,  
        void *p3 =NULL,  
        void *p4 =NULL,  
        void *p5 =NULL,  
        void *p6 =NULL,  
        void *p7 =NULL,  
        void *p8 =NULL,  
        void *p9 =NULL,  
        void *p10 =NULL,  
        void *p11 =NULL,  
        void *p12 =NULL,  
        void *p13 =NULL,  
        void *p14 =NULL,  
        void *p15 =NULL,  
        void *p16 =NULL,  
        void *p17 =NULL,  
        void *p18 =NULL,  
        void *p19 =NULL,  
        void *p20 =NULL,  
        void *p21 =NULL,  
        void *p22 =NULL,  
        void *p23 =NULL,  
        void *p24 =NULL,  
        void *p25 =NULL,  
        void *p26 =NULL,  
        void *p27 =NULL,  
        void *p28 =NULL,  
        void *p29 =NULL  
)
```

**Description:** A special version of fscanf that strips out any ' ' characters and replaces them with ' ' and converts the format string with ftoi so that %e and %f formats are converted properly for double precision.

**file (input) :** scan this file.

**format (input) :** use this printf style format.

**p0,p1,... (input) :** supply addresses of variables to save the results in.

### 2.4 getLine

```
int  
getLine( char s[], int lim)
```

**Description:** Read a line from standard input.

**s (input) :** char array in which to store the line

**lim (input) :** maximum number of chars that can be saved in s

### 2.5 getLine

```
int  
getLine( aString &answer )
```

**Description:** Read a line from standard input.

**s (input) :** char array in which to store the line

**lim (input) :** maximum number of chars that can be saved in s

## 2.6 ftoi

**aString**

**ftoi(const char \*ss)**

**Description:** "float to real aString conversion" This function is used to convert arguments to sscanf and fscanf so they work when OV\_USE\_DOUBLE is or is not defined. It will convert %e to %le and %f to %lf when OV\_USE\_DOUBLE is defined. Usually one should use sScanF and fScanF to have this done automatically so there is no need to call ftoi directly.

**ss (input) :** convert this string.

**author:** wdh

### 3 Constructing A++ Index Objects for Grid Functions: `getIndex`, `getBoundaryIndex`, `getGhostIndex`

A number of functions are provided to construct A++ Index objects for grid functions. The `getIndex` function constructs Index objects for the interior points of a grid function. The function `getBoundaryIndex` constructs the Index objects corresponding to a given boundary face. The `getGhostIndex` function constructs Index objects for a ghost line on a given face of the grid.

Recall that a gridFunction can be cell-centred or vertex-centred in any of the coordinate directions. Typically grid functions are either all vertex-centred, all cell-centred or else face-centred (a face-centred grid function is vertex-centred along one axis and cell-centred along the others). By passing a grid function to `getIndex`, `getBoundaryIndex` or `getGhostIndex` one can be sure to get the Index objects corresponding to the “centred-ness” of the grid function. This is important because the “interior” points of a vertex-centred grid function are different from the interior points of a cell-centred grid function or the interior points of a face-centred grid function. For grid functions that are `faceCenteredAll` (i.e. they have components that are `faceCentered` along each axis) there is a `getIndex` function with an extra argument that species which face centering axis to use.

#### 3.1 Index functions

Here are the `getIndex` functions (see also file `/users/henshaw/res/gf/OGgetIndex.h`). There are at least three flavours of each function. The first takes an `indexArray`, the second takes a grid function and component number and the third takes a grid function (in which case `component=0` is used to determine the Index’s). The function `getIndex` has an additional 2 flavours.

A summary of the arguments is as follows:

- `indexArray` : an intArray with dimensions (0:1,0:2), such as `indexRange`, `gridIndexRange` or `dimension`, that defines a region on the grid.
- `extra` : Increase the size of the domain by this many lines.
- `side`, `axis` : these arguments define which face to use
- `floatMappedGridFunction` : Use the cell-centeredness properties of this grid function to determine the index range. Versions of the function also exist for `doubleMappedGridFunction` and `intMappedGridFunction`.
- `component` : base the Index objects on this component of the grid function (required since different components of a grid function may be centred in different ways). The functions where the `component` argument is missing use `component=0`. The exception to the above rule is when the grid function is `faceCenteredAll` in which case `component =0,1` or `2` will indicate whether to return Index’s for the `faceCenteredAxis1`, `faceCenteredAxis2` or the `faceCenteredAxis3` components.
- `ghostLine` : return Index objects for this ghost line. Choose `ghostline=1` for the first ghost line, `ghostline=2` for the second ghost line. (`ghostline=0` will give the boundary).

#### 3.2 `getIndex` from an index array

```
void  
getIndex(const IntegerArray & indexArray,  
        Index & I1,  
        Index & I2,  
        Index & I3,  
        int extra1 =0,  
        int extra2 =OGgetIndexDefaultValue,  
        int extra3 =OGgetIndexDefaultValue)
```

**Description:** Return Index objects for the region defined by `indexArray`

**indexArray(0:1,0:2) (input):** defines a region

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1  
(so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.3 getIndex from a {fbat,double,int}MappedGridFunction

```
void
getIndex(const floatMappedGridFunction & u,
         Index & I1,
         Index & I2,
         Index & I3,
         int extra1,
         int extra2,
         int extra3 )
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component. To get Index's for a faceCenteredAll grid function use the getIndex function described next.

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1  
(so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

```
void
getIndex(const floatMappedGridFunction & u,
         int component,
         Index & I1,
         Index & I2,
         Index & I3,
         int extra1,
         int extra2,
         int extra3 )
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1  
(so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.4 getBoundaryIndex from an index array

```
void
getBoundaryIndex(const IntegerArray & indexArray,
                  int side,
                  int axis,
                  Index & Ib1,
```

```

Index & Ib2,
Index & Ib3,
int extra1 =0,
int extra2 =OGgetIndexDefaultValue,
int extra3 =OGgetIndexDefaultValue)

```

**Description:** return Index objects for a side of the region defined by indexArray

**indexArray(0:1,0:2) (input):** defines a region

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ib1,Ib2,Ib3 (output):** Index values for the given boundary of the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1  
(so that if you only set extra1=1 then by default extra2=extra3=1)

### 3.5 getBoundaryIndex from a {fbat,double,int}MappedGridFunction

```

void
getBoundaryIndex(const floatMappedGridFunction & u,
                  int side,
                  int axis,
                  Index & Ib1,
                  Index & Ib2,
                  Index & Ib3,
                  int extra1 =0,
                  int extra2 =OGgetIndexDefaultValue,
                  int extra3 =OGgetIndexDefaultValue)

```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1  
(so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.6 getBoundaryIndex from a {fbat,double,int}MappedGridFunction

```

void
getBoundaryIndex(const intMappedGridFunction & u,
                  int component,
                  int side,
                  int axis,
                  Index & Ib1,
                  Index & Ib2,
                  Index & Ib3,
                  int extra1,
                  int extra2,
                  int extra3
                  )

```

**Description:** return Index objects for a side of the region defined by indexArray

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function.

**component (input):** use this component of the grid function, UNLESS the grid function is `faceCenteredAll` in which case component =0,1 or 2 will indicate whether to return Index's for the `faceCenteredAxis1`, `faceCenteredAxis2` or the `faceCenteredAxis3` components.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ib1,Ib2,Ib3 (output):** Index values for the given boundary of the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

### 3.7 getGhostIndex from an index array

```
void  
getGhostIndex(const IntegerArray & indexArray,  
              int side,  
              int axis,  
              Index & Ig1,  
              Index & Ig2,  
              Index & Ig3,  
              int ghostLine =1,  
              int extra1 =0,  
              int extra2 =OGgetIndexDefaultValue,  
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line

**indexArray(0:1,0:2) (input):** defines a region

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline of the region

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

### 3.8 getGhostIndex from a {fbat,double,int}MappedGridFunction

```
void  
getGhostIndex(const floatMappedGridFunction & u,  
              int side,  
              int axis0,  
              Index & Ig1,  
              Index & Ig2,  
              Index & Ig3,  
              int ghostLine =1,  
              int extra1 =0,  
              int extra2 =OGgetIndexDefaultValue,  
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line of region defined by a grid function.

**u (input):** Base the Index's on the `indexRange` and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline on the given side

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

```
void  
getGhostIndex(const floatMappedGridFunction & u,  
              int component,  
              int side,  
              int axis,  
              Index & Ig1,  
              Index & Ig2,  
              Index & Ig3,  
              int ghostLine =1,  
              int extra1 =0,  
              int extra2 =OGgetIndexDefaultValue,  
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line of region defined by a grid function.

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline on the given side

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

extendedGridIndexRange

**IntegerArray**

**extendedGridIndexRange(const MappedGrid & mg)**

**Description:** Return the extendedGridIndexRange which is equal to mg.gridIndexRange except on interpolation boundaries where it is equal to mg.extendedIndexRange (i.e. it includes the ghost points). NOTE: Does not include ghost points on mixed physical/interpolation boundaries

**Author:** WDH

extendedGridRange

**IntegerArray**

**extendedGridRange(const MappedGrid & mg)**

**Description:** Return the extendedGridRange which is equal to mg.gridIndexRange except on interpolation boundaries AND mixedPhysicalInterpolation boundaries where it is equal to mg.extendedIndexRange (i.e. it includes the ghost points).

**Author:** WDH

### 3.9 getIndex from a {fbat,double,int}MappedGridFunction

```
void  
getIndex(const floatMappedGridFunction & u,  
        Index & I1,  
        Index & I2,  
        Index & I3,  
        int extra1,  
        int extra2,  
        int extra3 )
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component. To get Index's for a faceCenteredAll grid function use the getIndex function described next.

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

```
void  
getIndex(const floatMappedGridFunction & u,  
        int component,  
        Index & I1,  
        Index & I2,  
        Index & I3,  
        int extra1,  
        int extra2,  
        int extra3 )
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.10 getBoundaryIndex from a {fbat,double,int}MappedGridFunction

```
void  
getBoundaryIndex(const floatMappedGridFunction & u,  
                  int side,  
                  int axis,  
                  Index & Ib1,  
                  Index & Ib2,  
                  Index & Ib3,  
                  int extra1 =0,  
                  int extra2 =OGgetIndexDefaultValue,  
                  int extra3 =OGgetIndexDefaultValue)
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.11 getBoundaryIndex from a {fbat,double,int}MappedGridFunction

**void**

```
getBoundaryIndex(const intMappedGridFunction & u,
                 int component,
                 int side,
                 int axis,
                 Index & Ib1,
                 Index & Ib2,
                 Index & Ib3,
                 int extra1,
                 int extra2,
                 int extra3
               )
```

**Description:** return Index objects for a side of the region defined by indexArray

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function.

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ib1,Ib2,Ib3 (output):** Index values for the given boundary of the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

### 3.12 getGhostIndex from a {fbat,double,int}MappedGridFunction

**void**

```
getGhostIndex(const floatMappedGridFunction & u,
              int side,
              int axis0,
              Index & Ig1,
              Index & Ig2,
              Index & Ig3,
              int ghostLine =1,
              int extra1 =0,
              int extra2 =OGgetIndexDefaultValue,
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line of region defined by a grid function.

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline on the given side

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

```
void  
getGhostIndex(const floatMappedGridFunction & u,  
              int component,  
              int side,  
              int axis,  
              Index & Ig1,  
              Index & Ig2,  
              Index & Ig3,  
              int ghostLine =1,  
              int extra1 =0,  
              int extra2 =OGgetIndexDefaultValue,  
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line of region defined by a grid function.

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline on the given side

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

extendedGridIndexRange

**IntegerArray**

**extendedGridIndexRange(const MappedGrid & mg)**

**Description:** Return the extendedGridIndexRange which is equal to mg.gridIndexRange except on interpolation boundaries where it is equal to mg.extendedIndexRange (i.e. it includes the ghost points). NOTE: Does not include ghost points on mixed physical/interpolation boundaries

**Author:** WDH

extendedGridRange

**IntegerArray**

**extendedGridRange(const MappedGrid & mg)**

**Description:** Return the extendedGridRange which is equal to mg.gridIndexRange except on interpolation boundaries AND mixedPhysicalInterpolation boundaries where it is equal to mg.extendedIndexRange (i.e. it includes the ghost points).

**Author:** WDH

### 3.13 getIndex from a {fbat,double,int}MappedGridFunction

```
void  
getIndex(const floatMappedGridFunction & u,  
        Index & I1,  
        Index & I2,  
        Index & I3,  
        int extra1,  
        int extra2,  
        int extra3 )
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component. To get Index's for a faceCenteredAll grid function use the getIndex function described next.

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

```
void  
getIndex(const floatMappedGridFunction & u,  
        int component,  
        Index & I1,  
        Index & I2,  
        Index & I3,  
        int extra1,  
        int extra2,  
        int extra3 )
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.14 getBoundaryIndex from a {fbat,double,int}MappedGridFunction

```
void  
getBoundaryIndex(const floatMappedGridFunction & u,  
                  int side,  
                  int axis,  
                  Index & Ib1,  
                  Index & Ib2,  
                  Index & Ib3,  
                  int extra1 =0,  
                  int extra2 =OGgetIndexDefaultValue,  
                  int extra3 =OGgetIndexDefaultValue)
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.15 getBoundaryIndex from a {fbat,double,int}MappedGridFunction

**void**

```
getBoundaryIndex(const intMappedGridFunction & u,
                 int component,
                 int side,
                 int axis,
                 Index & Ib1,
                 Index & Ib2,
                 Index & Ib3,
                 int extra1,
                 int extra2,
                 int extra3
               )
```

**Description:** return Index objects for a side of the region defined by indexArray

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function.

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ib1,Ib2,Ib3 (output):** Index values for the given boundary of the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

### 3.16 getGhostIndex from a {fbat,double,int}MappedGridFunction

**void**

```
getGhostIndex(const floatMappedGridFunction & u,
              int side,
              int axis0,
              Index & Ig1,
              Index & Ig2,
              Index & Ig3,
              int ghostLine =1,
              int extra1 =0,
              int extra2 =OGgetIndexDefaultValue,
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line of region defined by a grid function.

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline on the given side

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

```
void  
getGhostIndex(const floatMappedGridFunction & u,  
              int component,  
              int side,  
              int axis,  
              Index & Ig1,  
              Index & Ig2,  
              Index & Ig3,  
              int ghostLine =1,  
              int extra1 =0,  
              int extra2 =OGgetIndexDefaultValue,  
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line of region defined by a grid function.

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline on the given side

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

extendedGridIndexRange

**IntegerArray**

**extendedGridIndexRange(const MappedGrid & mg)**

**Description:** Return the extendedGridIndexRange which is equal to mg.gridIndexRange except on interpolation boundaries where it is equal to mg.extendedIndexRange (i.e. it includes the ghost points). NOTE: Does not include ghost points on mixed physical/interpolation boundaries

**Author:** WDH

extendedGridRange

**IntegerArray**

**extendedGridRange(const MappedGrid & mg)**

**Description:** Return the extendedGridRange which is equal to mg.gridIndexRange except on interpolation boundaries AND mixedPhysicalInterpolation boundaries where it is equal to mg.extendedIndexRange (i.e. it includes the ghost points).

**Author:** WDH

### 3.17 getIndex from a {fbat,double,int}MappedGridFunction

```
void  
getIndex(const floatMappedGridFunction & u,  
        Index & I1,  
        Index & I2,  
        Index & I3,  
        int extra1,  
        int extra2,  
        int extra3 )
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component. To get Index's for a faceCenteredAll grid function use the getIndex function described next.

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

```
void  
getIndex(const floatMappedGridFunction & u,  
        int component,  
        Index & I1,  
        Index & I2,  
        Index & I3,  
        int extra1,  
        int extra2,  
        int extra3 )
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.18 getBoundaryIndex from a {fbat,double,int}MappedGridFunction

```
void  
getBoundaryIndex(const floatMappedGridFunction & u,  
                 int side,  
                 int axis,  
                 Index & Ib1,  
                 Index & Ib2,  
                 Index & Ib3,  
                 int extra1 =0,  
                 int extra2 =OGgetIndexDefaultValue,  
                 int extra3 =OGgetIndexDefaultValue)
```

**Description:**

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**I1,I2,I3 (output):** Index values for the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

### 3.19 getBoundaryIndex from a {fbat,double,int}MappedGridFunction

**void**

```
getBoundaryIndex(const intMappedGridFunction & u,
                 int component,
                 int side,
                 int axis,
                 Index & Ib1,
                 Index & Ib2,
                 Index & Ib3,
                 int extra1,
                 int extra2,
                 int extra3
               )
```

**Description:** return Index objects for a side of the region defined by indexArray

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function.

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ib1,Ib2,Ib3 (output):** Index values for the given boundary of the region

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

### 3.20 getGhostIndex from a {fbat,double,int}MappedGridFunction

**void**

```
getGhostIndex(const floatMappedGridFunction & u,
              int side,
              int axis0,
              Index & Ig1,
              Index & Ig2,
              Index & Ig3,
              int ghostLine =1,
              int extra1 =0,
              int extra2 =OGgetIndexDefaultValue,
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line of region defined by a grid function.

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function. Use the "first" component of the grid function, that is use the base value for each component.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline on the given side

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

**void**

```
getGhostIndex(const floatMappedGridFunction & u,
              int component,
              int side,
              int axis,
              Index & Ig1,
              Index & Ig2,
              Index & Ig3,
              int ghostLine =1,
              int extra1 =0,
              int extra2 =OGgetIndexDefaultValue,
              int extra3 =OGgetIndexDefaultValue)
```

**Description:** Get Index's corresponding to a given ghost-line of region defined by a grid function.

**u (input):** Base the Index's on the indexRange and cell-centredness associated with this grid function

**component (input):** use this component of the grid function, UNLESS the grid function is faceCenteredAll in which case component =0,1 or 2 will indicate whether to return Index's for the faceCenteredAxis1, faceCenteredAxis2 or the faceCenteredAxis3 components.

**side,axis (input):** defines which side=0,1 and axis=0,1,2

**Ig1,Ig2,Ig3 (output):** Index values for the given ghostline on the given side

**ghostline (input):** get Index's for this ghost line, can be positive, negative or zero. A value of zero would give the boundary, a value of 1 would give the first line outside and a value of -1 would give the first line inside.

**extra1,extra2,extra3 (input):** increase region by this many lines, by default extra1=0, while extra2 and extra3 default to extra1 (so that if you only set extra1=1 then by default extra2=extra3=1)

**Author:** WDH

extendedGridIndexRange

**IntegerArray**

**extendedGridIndexRange(const MappedGrid & mg)**

**Description:** Return the extendedGridIndexRange which is equal to mg.gridIndexRange except on interpolation boundaries where it is equal to mg.extendedIndexRange (i.e. it includes the ghost points). NOTE: Does not include ghost points on mixed physical/interpolation boundaries

**Author:** WDH

extendedGridRange

**IntegerArray**

**extendedGridRange(const MappedGrid & mg)**

**Description:** Return the extendedGridRange which is equal to mg.gridIndexRange except on interpolation boundaries AND mixedPhysicalInterpolation boundaries where it is equal to mg.extendedIndexRange (i.e. it includes the ghost points).

**Author:** WDH

## 4 OGFunction: A class for defining Twilight-zone flows

The class `OGFunction` and derived classes such as `OGTrigFunction`, `OGPolyfunction` and `OGPulseFunction` can be used to define “exact solutions” for PDE solvers in the manner that has come to be known as “twilight-zone flow” (Brown, 1985).

Basically this class defines an analytic function (such as a polynomial) and provides functions for evaluating this function and derivatives in convenient ways. Note that as of version 14 it is possible to evaluate any partial derivative using the `gd` (general derivative) member function.

It is often difficult to come up with exact solutions to test a PDE code. To overcome this difficulty one can add forcing functions to the PDE and boundary conditions that will make any given function an exact solution. For example given the time dependent PDE for vector function  $\mathbf{u}$

$$\mathbf{u}_t + a\mathbf{u}_x + b\mathbf{u}_y = \mathbf{f}(x, y, t)$$

with boundary conditions

$$B(\mathbf{u}) = \mathbf{g}(x, y, t)$$

one can make any given function  $\mathbf{U}(x, y, t)$  an exact solution by defining

$$\mathbf{f}(x, y, t) = \mathbf{U}_t + a\mathbf{U}_x + b\mathbf{U}_y$$

and

$$\mathbf{g}(x, y, t) = B(\mathbf{U}(x, y, t))$$

The class `OGPolyFunction` can be used to define the function  $\mathbf{U}(x, y, z, t)$  which is defined to be a polynomial in  $x, y, [z]$  and  $t$ . The class `OGTrigfunction` can be used to define a function  $\mathbf{U}(x, y, z, t)$  which is a trigonometric function such as  $\cos(2\pi x)\cos(2\pi y)\cos(2\pi t)$ .

The base class `OGFunction` does not define any functions and thus you should never construct one of these. You might, however, keep a pointer to the base class. This pointer can point to an object of a derived class.

### 4.1 Evaluate the function or a derivative at a point

```

real
operator()(const real x,
              const real y,
              const real z,
              const int n =0,
              const real t =0.)
```

  

```

real
operator()(const real x, const real y, const real z, const int n)
```

  

```

real
operator()(const real x, const real y, const real z)
```

  

```

real
x(const real x,
   const real y,
   const real z,
   const int n =0,
   const real t =0.)
```

**Description:** Evaluate the function or a derivative of the function at a point. The function name **x** can be replaced by any one of **t**, **x**, **y**, **z**, **xx**, **yy**, **zz**, **xy**, **xz**, **yz**, **xxx** or **xxxx**.

**x,y,z (input):** coordinates

**n (input):** component number (starting from 0)

**t (input):** time

```

real
gd(const int & ntd,
     const int & nxd,
     const int & nyd,
     const int & nzd,
     const real x0,
     const real y0,
     const real z0,
     const int n =0,
     const real t0 =0.)

```

**Description:** Evaluate a general derivative. The arguments are the same as in the corresponding **x** function except that the first 4 arguments specify the derivative to compute.

**ntd,nxd,nyd,nzd (input):** Specify the derivative to compute by indicating the order of each partial derivative.

**ntd** : number of time derivatives (order of the time derivative).  
**nxd** : number of x derivatives (order of the x derivative).  
**nyd** : number of y derivatives (order of the y derivative).  
**nzd** : number of z derivatives (order of the z derivative).

## 4.2 Evaluate the function or a derivative on a MappedGrid

**RealDistributedArray**

```

operator()(const MappedGrid & c, const Index & I1,
           const Index & I2, const Index & I3, const Index & N)
```

**RealDistributedArray**

```

operator()(const MappedGrid & c,
            const Index & I1,
            const Index & I2,
            const Index & I3,
            const Index & N,
            const real t =0.,
            const GridFunctionParameters::GridFunctionType & centering
            = defaultCentering)
```

**RealDistributedArray**

```

x(const MappedGrid & c,
   const Index & I1,
   const Index & I2, const
   Index & I3,
   const Index & N,
   const real t,
   const GridFunctionParameters::GridFunctionType & centering
   = defaultCentering)
```

**Description:** Evaluate the function or a derivative of the function at points on a MappedGrid. The function name **x** can be replaced by any one of **t**, **x**, **y**, **z**, **xx**, **yy**, **zz**, **xy**, **xz**, **yz**, **laplacian**, **xxx** or **xxxx**.

**I1,I2,I3 (input) :** Ranges that indicate points to use, for example by default the points

```
c.center()(I1, I2, I3, 0 : numberDimensions - 1)
```

are used.

**N (input) :** component indices to assign

**t (input) :** time

**centering (input):** This enum is found in `GridFunctionParameters`. It indicates the positions of the coordinates, one of

**defaultCentering** use the `c.center()` array (vertices for a vertex centered grid and cell centers for a cell-centered grid).

**vertexCentered** grid vertices, `c.vertex()`.

**cellCentered** `c.center()` for a cell-centered grid or else `c.corner()` for a vertex centered grid (the cell centers).

**faceCenteredAxis1** use the center of the cell face in the axis1 direction (defined by averaging the `c.vertex()` values for the y,z coordinates).

**faceCenteredAxis2** use the center of the cell face in the axis2 direction (defined by averaging the `c.vertex()` values for the x,z coordinates).

**faceCenteredAxis3** use the center of the cell face in the axis3 direction (defined by averaging the `c.vertex()` values for the x,y coordinates).

### RealDistributedArray

```
gd(const int & ntd,
    const int & nxn,
    const int & nyd,
    const int & nzd,
    const MappedGrid & c,
    const Index & I1,
    const Index & I2,
    const Index & I3,
    const Index & N,
    const real t0 =0.,
    const GridFunctionParameters::GridFunctionType & centering = defaultCentering)
```

**Description:** Evaluate a general derivative. The arguments are the same as in the corresponding `x` function except that the first 4 arguments specify the derivative to compute.

**ntd,nxd,nyd,nzd (input):** Specify the derivative to compute by indicating the order of each partial derivative.

**ntd** : number of time derivatives (order of the time derivative).

**nxd** : number of x derivatives (order of the x derivative).

**nyd** : number of y derivatives (order of the y derivative).

**nzd** : number of z derivatives (order of the z derivative).

### 4.3 Evaluate the function or a derivative on a CompositeGrid

```
realCompositeGridFunction
operator()(CompositeGrid & cg,
           const Index & N,
           const real t,
           const GridFunctionParameters::
           GridFunctionType & centering = defaultCentering)
```

```
realCompositeGridFunction
operator()(CompositeGrid & cg, const Index & N)
```

```
realCompositeGridFunction
operator()(CompositeGrid & cg)
```

```
realCompositeGridFunction
x(CompositeGrid & cg,
   const Index & N,
   const real t,
   const GridFunctionParameters::GridFunctionType & centering
   = defaultCentering)
```

**Description:** Evaluate the function or a derivative of the function at points on a CompositeGrid. The function name **x** can be replaced by any one of **t**, **x**, **y**, **z**, **xx**, **yy**, **zz**, **xy**, **xz**, **yz**, **laplacian**, **xxx** or **xxxx**.

**cg (input)** : use this grid. By default the points

```
c.center()(i1, I2, I3, 0 : numberOfDimensions - 1)
```

are used.

**N (input)** : evaluate these components

**t (input)** : time

**centering (input):** This enum is found in `GridFunctionParameters`. It indicates the positions of the coordinates, one of

**defaultCentering** use the `c.center()` array (vertices for a vertex centered grid and cell centers for a cell-centered grid).

**vertexCentered** grid vertices, `c.vertex()`.

**cellCentered** `c.center()` for a cell-centered grid or else `c.corner()` for a vertex centered grid (the cell centers).

**faceCenteredAxis1** use the center of the cell face in the axis1 direction (defined by averaging the `c.vertex()` values for the y,z coordinates).

**faceCenteredAxis2** use the center of the cell face in the axis1 direction (defined by averaging the `c.vertex()` values for the x,z coordinates).

**faceCenteredAxis3** use the center of the cell face in the axis1 direction (defined by averaging the `c.vertex()` values for the x,y coordinates).

## 4.4 OGPolyFunction

This class is derived from the OGFuction class and defines a function that is a polynomial in space times a polynomial in time of the form

$$U(x, y, z, n, t) = \left( \sum_{i,j,k} c(i, j, k, n) x^i y^j z^k \right) \sum_m a(m, n) t^m$$

Each component is a polynomial in space and time. The coefficient matrices,  $c(i, j, k, n)$  and  $a(m, n)$  can be specified or else default values can be used.

### 4.4.1 Constructor

```
OGPolyFunction(const int & degreeOfSpacePolynomial =2,
              const int & numberOfDimensions0 =3,
              const int & numberComponents0 =10,
              const int & degreeOfTimePolynomial =1)
```

**Description:** Create a polynomial with the given degree, number Of Space Dimensions and for a maximum number of components. The polynomial created is of the form

$U = (1 + x + x^2 + \dots + x^n)(1 + t + \dots + t^m)$ $U = (1 + x + x^2 + \dots + x^n + y + y^2 + \dots + y^n)(1 + t + \dots + t^m)$ $U = (1 + x + x^2 + \dots + x^n + y + y^2 + \dots + y^n + z + z^2 + \dots + z^n)(1 + t + \dots + t^m)$	in 1D in 2D in 3D
--	-------------------------

**degreeOfSpacePolynomial (input):** degree of the polynomial in x,y,z (n in the above formula).

**numberOfDimensions (input):** number of space dimensions, 1,2, or 3.

**numberOfComponents0 (input):** maximum number of components required.

**degreeOfTimePolynomial (input):** degree of the polynomial in t (m in the above formula).

**Notes:** Only polynomials with  $\text{degreeOfSpacePolynomial} < 5$  and  $\text{degreeOfTimePolynomial} < 5$  are supported

**Author:** WDH

### 4.4.2 setCoefficients

**void**

```
setCoefficients( const RealArray & c, const RealArray & a0 )
```

**Description:** Use this member function to set the coefficient matrices  $c$  and  $a$  of a *general* polynomial up to order 6.

$$U(x, y, z, n, t) = \left( \sum_{i,j,k} c(i, j, k, n) x^i y^j z^k \right) \sum_m a(m, n) t^m$$

Note that the values of `numberOfDimensions` and `numberOfComponents` given in the call to the constructor help to determine the polynomial created here.

**c (input) :** array of dimension  $c(0 : nx, 0 : nx, 0 : nx, 0 : \text{numberComponents} - 1)$  that gives the coefficients of the spatial polynomial of degree  $nx$  (`numberComponents` is the value given in call to the constructor). Some values in  $c$  may be ignored depending on the value for `numberOfDimensions`.

**a (input) :** array of dimension  $a(0 : nt, 0 : \text{numberComponents} - 1)$  that gives the coefficients of the time polynomial (`numberComponents` is the value given in call to the constructor).

**Author:** WDH

## 4.5 OGTrigFunction

### 4.5.1 Constructors

```
OGTrigFunction(const real & fx_=1.,
               const real & fy_=1.,
               const real & fz_=0.,
               const real & ft_=0.,
               const int & maximumNumberOfComponents =10)
```

**Description:** This class is derived from the OGFunction class and defines a function and defines a function that is a trigonometric polynomial:

$$u_n(x, y, z, t) = a(n) \cos(f_x(n)\pi(x - g_x(n))) \cos(f_y(n)\pi(y - g_y(n))) \cos(f_z(n)\pi(z - g_z(n))) \cos(f_t(n)\pi(t - g_t(n))) + c(n)$$

where  $a(n)$ ,  $f_x(n)$ ,  $f_y(n)$  etc. can be given values for each component n.

**fx\_, fy\_, fz\_, ft\_ (input):** give frequencies (constant for all components).

**maximumNumberOfComponents (input):** maximum number of components.

**Notes:** By default  $a(n) = 1$  and  $g_x(n) = g_y(n) = g_z(n) = g_t(n) = 0$ .

**Author:** WDH

```
OGTrigFunction(const RealArray & fx_,
                const RealArray & fy_,
                const RealArray & fz_,
                const RealArray & ft_)
```

**Description:** Use this constructor to supply different frequencies for different components.

**fx\_, fy\_, fz\_, ft\_ (input):** give frequencies for different components. The dimension of fx\_ will determine the maximumNumberOfComponents.

### 4.5.2 setAmplitudes

```
int
setAmplitudes(const RealArray & a_)
```

**Description:** Use this function to supply different amplitudes for different components.

**a\_ (input):** give amplitudes for different components. The dimension of a\_ should be equal to the maximumNumberOfComponents as determined by the call to the constructor.

### 4.5.3 setConstants

```
int
setConstants(const RealArray & c_)
```

**Description:** Use this function to supply different constants for different components.

**c\_ (input):** give constants for different components. The dimension of c\_ should be equal to the maximumNumberOfComponents as determined by the call to the constructor.

### 4.5.4 setFrequencies

```
int
setFrequencies(const RealArray & fx_,
               const RealArray & fy_,
               const RealArray & fz_,
               const RealArray & ft_)
```

**Description:** Use this function to supply different frequencies for different components.

**fx\_, fy\_, fz\_, ft\_ (input):** give frequencies for different components. The dimension of fx\_ will determine the maximumNumberOfComponents.

#### 4.5.5 setShifts

```
int  
setShifts(const RealArray & gx_,  
          const RealArray & gy_,  
          const RealArray & gz_,  
          const RealArray & gt_)
```

**Description:** Use this function to supply different shifts for different components.

**gx\_, gy\_, gz\_, gt\_ (input):** give shifts for different components. The dimensions of gx\_, gy\_,... should be equal to the maximumNumberOfComponents as determined by the call to the constructor.

## 4.6 OGPF

This class defines a *pulse* like function that can be useful for testing adaptive mesh refinement codes.

The pulse is defined as a generalized Gaussian,

$$u(\mathbf{x}, t) = a_0 \exp(-a_1 |\mathbf{x} - \mathbf{b}(t)|^{2p})$$

$$\mathbf{b}(t) = \mathbf{c}_0 + \mathbf{v}t$$

where  $p > \frac{1}{2}$ .

The derivatives of  $u$  are determined as follows. Letting

$$r = (x - b_0)^2 + (y - b_1)^2 + (z - b_2)^2$$

$$f = r^p$$

$$u = a_0 \exp(-a_1 f)$$

then

$$u_x = a_0 \exp(-a_1 f) [-a_1 f_x]$$

$$u_{xx} = a_0 \exp(-a_1 f) [(a_1 f_x)^2 - a_1 f_{xx}]$$

$$u_{xy} = a_0 \exp(-a_1 f) [a_1 f_x a_1 f_y - a_1 f_{xy}]$$

$$u_{xxxx} = a_0 \exp(-a_1 f) [-(a_1 f_x)^3 + 3f_x f_{xx} - f_{xxx}]$$

$$u_{xxxx} = a_0 \exp(-a_1 f) [(a_1 f_x)^4 + 3f_{xx}^2 - 6f_x^2 f_{xx} + 4f_x f_{xxx} - f_{xxxx}]$$

where

$$f_x = 2pr^{p-1}(x - b_0)$$

$$f_{xx} = 2pr^{p-1}(2(p-1)\frac{(x - b_0)^2}{r} + 1)$$

$$f_{xy} = 4p(p-1)r^{p-2}((x - b_0)(y - b_1))$$

$$f_{xxx} = 4p(p-1)r^{p-2}(x - b_0) \left[ (p-2)\frac{(x - b_0)^2}{r} + 3 \right]$$

$$f_{xxxx} = 4p(p-1)r^{p-2} \left[ 4(p-2)\frac{(x - b_0)^2}{r} ((p-3)\frac{(x - b_0)^2}{r} + 3) + 3 \right]$$

### 4.6.1 Constructor

```
OGPF(int numberOfDimensions_ = 2,
      int numberOfComponents_ = 1,
      real a0_ = 1.,
      real a1_ = 5.,
      real c0_ = 0.,
      real c1_ = 0.,
      real c2_ = 0.,
      real v0_ = 1.,
      real v1_ = 1.,
      real v2_ = 1.,
      real p_ = 1.)
```

**Description:** Define a pulse.

$$U = a_0 \exp(-a_1 |\mathbf{x} - \mathbf{b}(t)|^{2p})$$

$$\mathbf{b}(t) = \mathbf{c}_0 + \mathbf{v}t$$

**numberOfDimensions\_ (input):** number of space dimensions, 1,2, or 3.

**numberOfComponents\_ (input):** maximum number of components required.

**a0\_,a1\_,...,p\_ (input):** pulse parameters

#### 4.6.2 setRadius

```
int  
setRadius( real radius )
```

**Description:** Set the approximte radius of the pulse. This will set the parameter  $a_1$  according to the formula  $\text{radius} = 1/\sqrt{a_1}$ .

**radius (input):** approximate radius.

**Author:** WDH

#### 4.6.3 setRadius

```
int  
setShape( real p_ )
```

**Description:** Set the *shape* parameter  $p$ .  $p = 1$  gives a Gaussian pulse, choosing a larger value of  $p$  will cause the pulse to flatten on the top and approach a top-hat function as  $p$  tends to infinity.

**p\_ (input):** shape parameter,  $p > \frac{1}{2}$ .

**Author:** WDH

#### 4.6.4 setCentre

```
int  
setCentre( real c0_=0.,  
           real c1_=0.,  
           real c2_=0.)
```

**Description:** Set the pulse centre.

**c0\_,c1\_,c2\_ (input):** centre.

**Author:** WDH

#### 4.6.5 setVelocity

```
int  
setVelocity( real v0_=1.,  
             real v1_=1.,  
             real v2_=1.)
```

**Description:** Set the pulse velocity.

**v0\_,v1\_,v2\_ (input):** velocity.

**Author:** WDH

#### 4.6.6 setCoefficients

```
void  
setParameters( int numberDimensions_=2,  
               int numberComponents_=1,  
               real a0_=1.,  
               real a1_=5.,  
               real c0_=0.,  
               real c1_=0.,  
               real c2_=0.,  
               real v0_=1.,  
               real v1_=1.,  
               real v2_=1.,  
               real p_=1.)
```

**Description:** Use this member function to set parameters.

Define a pulse.

$$U = a_0 \exp(-a_1 |\mathbf{x} - \mathbf{c}(t)|^p) \quad p > \frac{1}{2}$$
$$\mathbf{c}(t) = \mathbf{c}_0 + \mathbf{v}t$$

**numberOfDimensions\_ (input):** number of space dimensions, 1,2, or 3.

**numberOfComponents\_ (input):** maximum number of components required.

**a0\_,a1\_,...,p\_ (input):** pulse parameters.

**Author:** WDH

## 4.7 Examples

The file `Overture/tests/tz.C` is an example code that uses the `OGPolyFunction`, `OGTrigFunction` and `OGPulseFunction` classes. See also the examples in the primer directory.

## 5 Data-base Access Functions

Here are the functions that can be used to read in various objects from a data base.

### 5.1 getFromADataBase(CompositeGrid & cg,...)

```
int  
getFromADataBase(CompositeGrid & cg,  
                  const aString & fileName,  
                  const aString & gridName =nullString,  
                  const bool & checkTheGrid =FALSE)
```

**Description:** Read in a CompositeGrid from a data-base file. The data-base file can either have been generated by the Overture grid generator **ogen** or by **Xcog** or **Chalmesh** from Anders Petersson. To read in a grid from a previous version of Overture you may have to use the **decompress** function (see below).

**fileName (input) :** name of the data-base file. Currently only HDF data-base files are supported. This routine will search for both fileName and fileName.hdf

**gridName (input) :** optional name for the grid. For example, this is the name given to the grid if it was created with ogen. If no gridName is supplied then the first CompositeGrid found in the file will be used.

**checkTheGrid (input) :** If TRUE the grid is checked for consistency such as valid interpolation points etc. by calling the function **checkOverlappingGrid**. It might be wise to use this option if you are reading a grid from Xcog or Chalmesh.

**Return values:** 0=success, 1=unable to open the file.

**Remark:** By default a data-base file made with one version of Overture will not be compatible with a newer version of Overture. The reason for this is that the data-base file is stored in a compressed way that makes it smaller in size and faster to read/write. There is a mechanism for converting data-base files containing grids from one version of Overture to another. The program **Overture/bin/decompress** will create a decompressed version of the grid file which can be read (more or less correctly) by new versions of Overture. For example, to convert from v18 to v19 you would run Overture.v18/bin/decompress on an overlapping grid built from v18. The resulting grid file can be read by Overture.v19. If you want to use this for v17 files you would have to copy the file Overture.v18/bin/decompress.C and compile it with v17.

## 6 Display functions for arrays (writing arrays to files)

### 6.1 display: display an A++ array

```
int  
display( const floatArray & x, const char *label, const char *format_ )
```

**Description:** Another version of display – pass a format but no FILE

**x (input):** array to display. There are also versions of this routine for int and double arrays.

**label (input):** optional header label

**format (input):** an optional format such as "%6.1e" or "%11.4e" (note blank at end) that will be used to display each element in the array. The default is "%11.4e"

### 6.2 display: save an A++ array in a file

```
int  
display( const floatArray & x,  
         const char *label = NULL,  
         FILE *file = NULL,  
         const char *format_ = NULL)
```

**Description:** Display an A++ array

**x (input):** array to display. There are also versions of this routine for int and double arrays.

**label (input):** optional header label

**file (input):** optionally supply a file to print to.

**format (input):** an optional format such as "%6.1e" or "%11.4e" (note blank at end) that will be used to display each element in the array. The default is "%11.4e"

### 6.3 display an A++ array with DisplayParameters

```
int  
display( const floatArray & x, const char *label, const DisplayParameters & displayParameters )
```

**Description:** Another version of display – pass a format but no FILE

**x (input):** array to display. There are also versions of this routine for int and double arrays.

**label (input):** optional header label

**format (input):** an optional format such as "%6.1e" or "%11.4e" (note blank at end) that will be used to display each element in the array. The default is "%11.4e"

**displayParameters (input):** provide parameters for display.

### 6.4 displayMask

```
int  
displayMask( const intArray & mask,  
             const aString & label = nullString,  
             FILE *file = NULL)
```

**Description:** Display the mask array in a MappedGrid in a reasonable way. The mask array in a MappedGrid is a bit-mapping that is difficult to look at if displayed in the formal way. This routine will display the mask in a more compact form (although some information is not printed) where each entry printed will mean:

1 : ISdiscretizationPoint

2 : ISghostPoint

-1 : ISinterpolationPoint

## 7 Integrate: integrate grid functions on overlapping grids

The `Integrate` class has functions that can be used to integrate a grid function over a domain or over the boundary (or a subset of the boundary). For example, one may want to compute the total mass found in a domain or compute the force on a body.

Integrating a function on an overlapping grid is non-trivial since care must be taken in the region where grids overlap.

The most important member functions are

**volumeIntegral(u)** : compute the volume integral of a `RealCompositeGridFunction` u.

**surfaceIntegral(u)** : compute the surface integral of a `RealCompositeGridFunction` u.

**defineSurface(s,...)** : define a sub-surface ‘s’ as a collection of sides of grids. For example, for the ‘sphere in a box’ grid you could define a sub-surface that represents the two surface grids on the sphere.

**surfaceIntegral(u,s)** : compute the surface integral on the surface ‘s’.

To use the `Integrate` class you should follow the example given below. (file `Overture/examples/ti.C`)

### 7.1 Member Functions

#### 7.2 constructor

**Integrate()**

**Description:** Default constructor.

#### 7.3 constructor

**Integrate(CompositeGrid & cg\_)**

**Description:**

**cg\_ (input)** : supply a grid on which to integrate.

#### 7.4 updateToMatchGrid

**int**

**updateToMatchGrid( CompositeGrid & cg\_ )**

**Description:** Call this routine to supply a grid or to indicate that the grid has changed.

**cg (input)** : supply a grid on which to integrate.

#### 7.5 defineSurface

**int**

**defineSurface(const int & surfaceNumber, const int & numberOfFaces\_, IntegerArray & boundary )**

**Description:** Specify the sides of grids that define a ”surface”. A surface represents some subset of the boundary of an entire domain. For example, for the sphere-in-a-box grid a surface could represent the surface of the sphere. To define a surface you must supply:

**surfaceNumber (input)** : a surface identifier. This value must be bigger than or equal to zero. Normally surfaces should be numbered starting from zero.

**numberOfFaces (input)** : the number of faces that make up the surface.

**boundary (input):** `boundary(3,numberofFaces) : (side,axis,grid)=boundary(0:2,i)`  $i=0,1,\dots,numberofFaces$ . To define a surface you must supply a list of sides of grids.

## 7.6 `numberOfFacesOnASurface`

```
int  
numberOfFacesOnASurface(int surfaceNumber) const
```

**Description:** Return the number of faces that form a given surface

**Return value:** number of faces for surface.

## 7.7 `getBoundaryDefinition`

```
const BodyDefinition &  
getBodyDefinition() const
```

**Description:** Return the BodyDefinition object which defines the relationship between grids and boundaries.

## 7.8 `getFace`

```
int  
getFace(int surfaceNumber,int face, int & side, int & axis, int & grid) const
```

**Description:** Return the data for a particular face of a surface.

**side,axis,grid (output):** this face corresponds to these values.

**Return value:** 0 for success.

## 7.9 `integrationWeights`

```
RealCompositeGridFunction &  
integrationWeights()
```

**Description:** Return the integration weights.

**Return value:** a grid function that holds the integration weights.

## 7.10 `leftNullVector`

```
RealCompositeGridFunction &  
leftNullVector()
```

**Description:** Return the left null vector of the Neumann problem. This vector is related to the integration weights.

**Return value:** a grid function that holds the left null vector.

## 7.11 `surfaceIndex`

```
int  
surfaceIndex( int surfaceNumber )
```

**Access level:** protected.

**Description:** For a given surfaceNumber determine the surfaceIndex such that surfaceIdentifier(surfaceIndex)==surfaceNumber.  
Return -1 if no match is found.

**surfaceNumber (input) :** the surface ID for a user defined surface.

**Return value:** the index into the surfaceIdentifier array, or -1 if no match exists.

## 7.12 surfaceIntegral

**real**

**surfaceIntegral(const RealCompositeGridFunction & u, const int & surfaceNumber = -1)**

**Description:** Compute the surface integral of u.

**u (input) :** function to integrate. This function must be defined at the appropriate points.

**surfaceNumber (input) :** the surface identifier as defined through a call to `defineSurface`. If no surfaceNumber is specified then the entire surface will be integrated.

**Return value:** The integral of u.

**Author:** WDH

## 7.13 surfaceIntegral

**int**

**surfaceIntegral(const RealCompositeGridFunction & u,  
const Range & C,  
RealArray & integral,  
const int & surfaceNumber = -1)**

**Description:** Compute the surface integral of u, one or more components.

**u (input) :** function to integrate. This function must be defined at the appropriate points.

**C (input) :** integrate these components.

**integral (output):** array of values, `integral(C)`, the integrals of the components.

**surfaceNumber (input) :** the surface identifier as defined through a call to `defineSurface`. If no surfaceNumber is specified then the entire surface will be integrated.

**Author:** WDH

## 7.14 volumeIntegral

**real**

**volumeIntegral( const RealCompositeGridFunction & u )**

**Description:** Compute the volume integral of u.

**u (input) :** function to integrate. This function must be defined at the appropriate points.

**Author:** WDH

## 8 TridiagonalSolver: Solve sets of tridiagonal systems or block tridiagonal systems

The TridiagonalSolver class can be used to solve tridiagonal systems or block tridiagonal systems. Sets of tridiagonal systems can be solved such as those that are formed when line smoothing is performed on a 2 or 3 dimensional grid or an ADI type method is used to solve a PDE. Currently only blocks or size 2 or 3 are implemented.

The two basic steps to solve a tridiagonal system are to first factor the system by calling the `factor` member function and then solve the system using `solve`.

There are three types of boundary conditions supported, `normal`, `extended` and `periodic`. A `normal` matrix is of the form

$$\text{normal} = \begin{bmatrix} b_0 & c_0 & & & \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & c_2 & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix}$$

The extended matrix allows extra entries on the first and last rows (required by some PDE boundary conditions)

$$\text{extended} = \begin{bmatrix} b_0 & c_0 & a_0 & & \\ a_1 & b_1 & c_1 & & \\ a_2 & b_2 & c_2 & & \\ \ddots & \ddots & \ddots & \ddots & \\ a_{n-1} & b_{n-1} & c_{n-1} & & \\ c_n & a_n & b_n & & \end{bmatrix}$$

The periodic case is

$$\text{periodic} = \begin{bmatrix} b_0 & c_0 & & & a_0 \\ a_1 & b_1 & c_1 & & \\ a_2 & b_2 & c_2 & & \\ \ddots & \ddots & \ddots & & \\ a_{n-1} & b_{n-1} & c_{n-1} & & \\ c_n & a_n & b_n & & \end{bmatrix}$$

Here is an example code that shows how to use the TridiagonalSolver (file `Overture/tests/trid.C`)

### 8.1 Member Functions

### 8.2 constructor

#### TridiagonalSolver()

**Description:** Use this class to solve a tridiagonal system. The system may be block tridiagonal. There may be multiple independent tridiagonal systems to be solved. The basic tridiagonal system is (`type=normal`)

$$A = \begin{vmatrix} b[0] & c[0] & & & \\ a[1] & b[1] & c[1] & & \\ & a[2] & b[2] & c[2] & \\ & & \ddots & \ddots & \\ & & a[.] & b[.] & c[.] \\ & & & a[n] & b[n] \end{vmatrix}$$

We can also solve the `type=periodic`

$$A = \begin{vmatrix} b[0] & c[0] & & a[0] \\ a[1] & b[1] & c[1] & \\ & a[2] & b[2] & c[2] \\ & & \ddots & \ddots \\ & & a[.] & b[.] & c[.] \\ & c[n] & & a[n] & b[n] \end{vmatrix}$$

and the `type=extended`

$$A = \begin{vmatrix} b[0] & c[0] & a[0] \\ a[1] & b[1] & c[1] \\ a[2] & b[2] & c[2] \\ & \ddots & \ddots \\ & a[.] & b[.] & c[.] \\ c[n] & a[n] & b[n] \end{vmatrix}$$

which may occur with certain boundary conditions.

This class expects the matrices a,b,c to be passed separately and to be of the form

- $a(I1,I2,I3)$ ,  $b(I1,I2,I3)$ ,  $c(I1,I2,I3)$  : if the block size is 1.
- $a(b,b,I1,I2)$  : if the block size is  $b > 1$ .

The ‘axis’ argument to the member functions indicates which of  $I1, I2$  or  $I3$  represents the axis along which the tridiagonal matrix extends. The other axes can be used to hold independent tridiagonal systems. Thus if `axis=0` then  $a(i1,i2,i3)$   $i1=0,1,2,\dots,n$  are the entries in the tridiagonal matrix for each fixed  $i2$  and  $i3$ . If `axis=1` then  $a(i1,i2,i3)$   $i2=0,1,2,\dots,n$  are the entries in the tridiagonal matrix for fixed  $i1$  and  $i3$ .

### 8.3 factor

```
int
factor(realArray & a_,
       realArray & b_,
       realArray & c_,
       const SystemType & type_=normal,
       const int & axis_=0,
       const int & block_=1)
```

**Description:** Factor the tri-diagonal (block) matrix defined by (a,b,c). NOTE: This routine keeps a reference to (a,b,c) and factors in place.

**a,b,c (input/output):** on input the 3 diagonals, on output the LU factorization

**type (input):** One of `normal`, `periodic` or `extended`.

**axis (input):** 0, 1, or 2. See the comments below.

**block (input):** block size. If `block=2` or `3` then the matrix is block tridiagonal.

**Notes:** This class expects the matrices a,b,c to be of the form

- $a(I1,I2,I3)$ ,  $b(I1,I2,I3)$ ,  $c(I1,I2,I3)$  : if the block size is 1.
- $a(b,b,I1,I2)$ ,  $b(b,b,I1,I2)$ ,  $c(b,b,I1,I2)$  : if the block size is  $b > 1$ .

The ‘axis’ argument to the member functions indicates which of  $I1, I2$  or  $I3$  represents the axis along which the tridiagonal matrix extends. The other axes can be used to hold independent tridiagonal systems. Thus if `axis=0` then  $a(i1,i2,i3)$   $i1=0,1,2,\dots,n$  are the entries in the tridiagonal matrix for each fixed  $i2$  and  $i3$ . If `axis=1` then  $a(i1,i2,i3)$   $i2=0,1,2,\dots,n$  are the entries in the tridiagonal matrix for fixed  $i1$  and  $i3$ .

## 8.4 solve

```
real  
sizeOf( FILE *file =NULL) const
```

**Description:** Return number of bytes allocated by Oges; optionally print detailed info to a file

**file (input) :** optionally supply a file to write detailed info to. Choose file=stdout to write to standard output.

**Return value:** the number of bytes.

## 8.5 solve

```
int  
solve(const realArray & r_ this is not really const,  
      const Range & R1 =nullRange,  
      const Range & R2 =nullRange,  
      const Range & R3 =nullRange)
```

**Description:** Solve a set of tri-diagonal systems.

**r\_ (input/output) :** rhs vector on input, solution on output. This is declared const to avoid compiler warnings.

**R1,R2,R3:** Specifies which systems to solve. By default all the systems are solved. These Ranges must be a subset of the collection of systems that are found in the matrices passed to the factor function. One of these is arguments is ignored, the one corresponding to the axis along which the tridiagonal system extends.

## 9 FortranIO : Write Fortran formatted or unformatted files from C++

Use this class to write fortran files from C++.

### 9.1 constructor

**FortranIO()**

**Description:** Build a FortranIO object.

### 9.2 open

**int**

**open(const aString & fileName,  
      const aString & fileForm,  
      const aString & fileStatus,  
      const int & fortranUnitNumber =25)**

**Description:** Open a fortran file with a fortran statement of the form:

```
open (unit=io, file=fileName,form=fileForm,status=fileStatus)
```

**fileName (input) :** name of the file.

**fileForm (input) :** a valid fortran file format such as "formatted" or "unformatted". \*\*\* only "unformatted" is currently supported.

**fileStatus (input) :** a valid file status such as "old", "new", "unknown"

**fortranUnitNumber (input) :** a positive integer.

### 9.3 close

**int**

**close()**

**Description:** Close a fortran file.

### 9.4 print( int )

**int**

**print(const int & i)**

**Description:** Save an int in the file.

### 9.5 print( float )

**int**

**print(const float & f)**

**Description:**

### 9.6 print( double )

**int**

**print(const double & d)**

**Description:**

## **9.7 print( int\* )**

```
int  
print(const int *a, const int & count)
```

**Description:** Save an array of values.

## **9.8 print( fbat\* )**

```
int  
print(const float *a, const int & count)
```

**Description:** Save an array of values.

## **9.9 print( double\* )**

```
int  
print(const double *a, const int & count)
```

**Description:** Save an array of values.

## **9.10 print( aString )**

```
int  
print(const aString & label)
```

**Description:**

## **9.11 print( intArray )**

```
int  
print(const intArray & u)
```

**Description:**

## **9.12 print( fbatArray )**

```
int  
print(const floatArray & u)
```

**Description:**

## **9.13 print( doubleArray )**

```
int  
print(const doubleArray & u)
```

**Description:**

## **9.14 print( intArray,fbatArray )**

```
int  
print(const intArray & u, const floatArray & v)
```

**Description:** Output an int and float array.

## **9.15 print( intArray,doubleArray )**

```
int  
print(const intArray & u, const doubleArray & v)
```

**Description:**

## **9.16 read( int )**

```
int  
read(const int & i)
```

**Description:** Save an int in the file.

## **9.17 read( fbat )**

```
int  
read(const float & f)
```

**Description:**

## **9.18 read( double )**

```
int  
read(const double & d)
```

**Description:**

## **9.19 read( int\* )**

```
int  
read(const int *a, const int & count)
```

**Description:** Save an array of values.

## **9.20 read( fbat\* )**

```
int  
read(const float *a, const int & count)
```

**Description:** Save an array of values.

## **9.21 read( double\* )**

```
int  
read(const double *a, const int & count)
```

**Description:** Save an array of values.

## **9.22 read( aString )**

```
int  
read(const aString & label)
```

**Description:**

## **9.23 read( intArray )**

```
int  
read(const intArray & u)
```

**Description:** Read in an array – the array must be dimensioned to the correct size.

## **9.24 read( fbatArray )**

```
int  
read(const floatArray & u)
```

**Description:** Read in an array – the array must be dimensioned to the correct size.

## **9.25 read( doubleArray )**

**int**

**read(const doubleArray & u)**

**Description:** Read in an array – the array must be dimensioned to the correct size.

# 10 Reference Counted Objects

## 10.1 Introduction

This section describes our notion of reference counted objects and describes the class `List1OfReferenceCountedObjects`.

Grids, grid functions and A++ arrays are all reference counted objects.

A reference counted object can be extremely useful for a number of reasons

- a reference counted objects can be easily shared between different classes without the need to use pointers.
- a reference counted object can automatically delete itself when it's reference count goes to zero.
- Someone who makes a reference to a reference counted object does not have to worry that the object will be deleted by someone else.

Any object can be made into a reference counted object by supplying certain member functions and by following certain rules. We will describe how to do this later in this section.

A reference counted object extends the notion of a reference in C++. The statement

```
realArray & u = v;
```

makes u become a reference (alias) for v. Similiarly the statements

```
realArray u;  
u.reference(v);
```

also make u a reference for v; In both cases changing u with a statement like `u=5.`; will cause v to also change. However, the latter method for creating a reference is more dynamic. For example, at a later stage u can reference a different array, `u.reference(w)` or u can break the reference `u.breakReference()`. When a reference is broken u will be given it's own copy of the array data.

## 10.2 How to Write a Reference Counted Class

In this section we describe how to write a reference counted class. Examples of reference counted classes are `MappedGrid`, `GridCollection`, `typeMappedGridFunction` and `typeGridCollectionFunction`.

Here is an example of a header file for a reference counted class (file `Overture/examples/ReferenceCountedClass.h`)

```
1  /* -*-Mode: c++; -*- */  
2  #ifndef REFERENCE_COUNTED_CLASS  
3  #define REFERENCE_COUNTED_CLASS "ReferenceCountedClass.h"  
4  
5  //=====  
6  // ReferenceCountedClass  
7  //  
8  // This class demonstrates a simple example of a reference counted class  
9  //  
10 //=====  
11 #include "ReferenceTypes.h"           // define intR floatR doubleR  
12 typedef floatR realR;  
13 typedef float  real;  
14  
15 class ReferenceCountedClass : public ReferenceCounting    // derive the class from ReferenceCounting  
16 {  
17     public:  
18         intR i;                                // this is a reference counted int  
19         realR x;                               // this is a reference counted real  
20         intArray array;                         // A++ arrays are reference counted  
21  
22     ReferenceCountedClass( );                  // default constructor  
23     ~ReferenceCountedClass();                 // destructor  
24     ReferenceCountedClass(const ReferenceCountedClass & rcc,          // copy constructor  
25                            const CopyType copyType = DEEP );  
26     ReferenceCountedClass& operator=( const ReferenceCountedClass & rcc ); // assignment operator  
27     void reference( const ReferenceCountedClass & rcc );                // reference this object to another  
28     void breakReference();                   // break a reference
```

```

29
30 private:
31     void initialize();                                     // used by constructors
32     // These are used by list's of ReferenceCounting objects
33     virtual void reference( const ReferenceCounting & rcc )
34     { ReferenceCountedClass::reference( (ReferenceCountedClass&) rcc ); }
35     virtual ReferenceCounting & operator=( const ReferenceCounting & rcc )
36     { return ReferenceCountedClass::operator=( (ReferenceCountedClass&) rcc ); }
37     virtual ReferenceCounting* virtualConstructor( const CopyType ct = DEEP )
38     { return ::new ReferenceCountedClass(*this,ct); }
39
40 protected:
41     class RCData : public ReferenceCounting    // this class hold the reference counted data
42     {
43     public:
44         int i;                                         // here is where i is really kept
45         real x;                                       // here is where x is really kept
46         RCData();
47         ~RCData();
48         RCData& operator=(const RCData & rcc );
49     private:
50         // These are used by list's of ReferenceCounting objects
51         virtual void reference( const ReferenceCounting & rcc )
52         { RCData::reference( (RCData&) rcc ); }
53         virtual ReferenceCounting & operator=( const ReferenceCounting & rcc )
54         { return RCData::operator=( (RCData&) rcc ); }
55         virtual ReferenceCounting* virtualConstructor( const CopyType )
56         { return ::new RCData(); }
57     };
58 protected:
59     RCData *rcData;
60 };
61 #endif

```

A reference counted class should

- be derived from the ReferenceCounting class.
- contain a copy constructor which can be a deep or shallow copy.
- have an assignment operator which is a deep copy.
- have a `reference` function
- have a `breakReference` function
- create another class to hold all the data associated with the class (class RCData in the example).
- contain a pointer to a class that holds the data for the object
- a private section with functions `reference`, `assignment`, and `virtualConstructor`, see example. These member functions allow all class's that are derived from the ReferenceCounting class to be put in a list.

Here is the implementation of ReferenceCountedClass (file Overture/examples/ReferenceCountedClass.C)

```

1 //=====
2 //      This file defines the functions for the ReferenceCountedClass Class
3 //
4 //=====
5
6 #include "ReferenceCountedClass.h"
7
8 //=====
9 // Default constructor
10 //=====
11 ReferenceCountedClass::
12 ReferenceCountedClass ()
13 {
14 }
15

```

```

16 //=====
17 // Copy constructor, deep copy by default
18 //=====
19 ReferenceCountedClass::
20 ReferenceCountedClass( const ReferenceCountedClass & rcc, const CopyType copyType )
21 {
22     if( copyType==DEEP )
23     {
24         initialize();           // put "this" object into a valid state
25         (*this)=rcc;          // this is a deep copy
26     }
27     else
28     {
29         rcData=rcc.rcData;    // set pointer to reference counted data
30         rcData->incrementReferenceCount();
31         reference(rcc);      // reference this object to rcc
32     }
33 }
34
35 //=====
36 // Destructor
37 //=====
38 ReferenceCountedClass::
39 ~ReferenceCountedClass ()
40 {
41     if( rcData->decrementReferenceCount() == 0 )   // if there are no references, then
42         delete rcData;                            // delete the reference counted data
43 }
44
45 //=====
46 // Assign initial values to variables
47 //=====
48 void ReferenceCountedClass::
49 initialize()
50 {
51     rcData = new ReferenceCountedClass::RCData;        // create a reference counted data object
52     rcData->incrementReferenceCount();
53
54     i.reference(rcData->i);                         // make reference counted i reference the "true" i
55     x.reference(rcData->x);                         // make reference counted x reference the "true" x
56     array.redim(3);
57 }
58
59 //=====
60 // Reference this object to another
61 //=====
62 void ReferenceCountedClass::
63 reference( const ReferenceCountedClass & rcc )
64 {
65     if( this==&rcc ) // no need to do anything in this case
66     return;
67     if( rcData->decrementReferenceCount() == 0 )
68         ::delete rcData;
69     rcData=rcc.rcData;
70     rcData->incrementReferenceCount();
71     i.reference(rcc.i);
72     x.reference(rcc.x);
73     array.reference(rcc.array);
74 }
75
76 //=====
77 // break the reference that this object has with any other objects
78 // after calling this function the object will have a separate copy of all member data
79 //=====
80 void ReferenceCountedClass::
81 breakReference()
82 {
83     // If there is only 1 reference, no need to make a new copy
84     if( rcData->getReferenceCount() != 1 )
85     {
86         ReferenceCountedClass rcc = *this; // makes a deep copy
87         reference(rcc);                // make a reference to this new copy

```

```

88     }
89 }
90
91 //=====
92 // Assignment with = is a deep copy
93 //=====
94 ReferenceCountedClass & ReferenceCountedClass::operator= ( const ReferenceCountedClass & rcc )
95 {
96     *rcData=*rcc.rcData;      // deep copy of reference counted data
97     i    =rcc.i;             // deep copy of the member data
98     x    =rcc.x;
99     array=rcc.array;
100    return *this;
101 }
102
103
104
105 //=====
106 // Default constructor for the reference counted data
107 //=====
108 ReferenceCountedClass::RCData::RCData()
109 {
110 }
111
112
113 //=====
114 // Default destructor for the reference counted data
115 //=====
116 ReferenceCountedClass::RCData::~RCData()
117 {
118 }
119
120
121 //=====
122 // Assignment with = is a deep copy
123 //=====
124 ReferenceCountedClass::RCData::operator=(const ReferenceCountedClass::RCData & rcc )
125 {
126     i=rcc.i;
127     x=rcc.x;
128 }
129
130

```

A reference counted class should be derived from the ReferenceCounting class (file /ReferenceCounting.h)

### 10.3 Class ListOfReferenceCountedObjects

This is a template list class for holding holding reference counted objects.

The class is declared as

```
template<class T>
class ListOfReferenceCountedObjects : public ReferenceCounting
```

so that T is the generic name of the class whose instances will be placed in the list.

#### 10.3.1 Constructors

DifferentialAndBoundaryOperators( )	Default constructor
ListOfReferenceCountedObjects()	default constructor
ListOfReferenceCountedObjects(int numberofElements)	Create a list with a given number of elements

#### 10.3.2 Public Member Functions

Here are the public member functions.

ListOfReferenceCountedObjects& operator=(ListOfReferenceCountedObjects&)	
void addElement(int index)	Add an object to the list
void addElement()	Add an object to the end of the list
void addElement(T & t, int index)	Add an object and reference to t
void addElement(T & t )	Add an object to the end, and reference to t
int getLength()	Get length of list.
T& operator[](int index)	Reference the object at a given location.
void deleteElement(T & X)	Find an element on the list and delete it
void deleteElement(int index)	Delete the element at the given index
void deleteElement()	Delete the element appearing last in the list
void swapElements(int i, int j)	Swap two elements (for sorting)
int getIndex(T & X)	get the index for an element
void reference(ListOfReferenceCountedObjects<T> & list )	reference one list to another
void breakReference()	break any references with this list

### 10.3.3 Examples

This class supports two types of reference counting. The suggested way to do reference counting is to use the reference and breakReference member functions as shown in the examples below. If instead you want to keep pointers to ListOfReferenceObjects then you can use the incrementReferenceCount(), decrementReferenceCount() and getReferenceCount() member functions (derived from the ReferenceCounting in order to mange references). With this latter mode of reference counting it is up to you to call delete when the reference count reaches zero.

Typical Usage:

```

floatArray a(10),b(5);
ListOfReferenceCountedObjects<floatArray> list,list2;
list.addElement();           // add an element
list[0].reference(a);      // reference to array a
list.addElement(b);         // add element and reference to b in one step
list[1]=b;
list[0]=1.;                 // same as a=1.;

list2.reference(list);     // list2 and list are now the same
list2[0]=2.;               // same as list[0]=2.;
list2.breakReference();
list2[0]=5.;               // does not change list[0]

```

# Index

block tridiagonal solver, 39

data base access functions, 34

data-base files

conversion to new versions, 34

decompress, 34

display

of A++ arrays, 35

Fortarn

write fortran files from C++, 42

fScanF, 8

ftor, 9

getBoundaryIndex, 10

getGhostIndex, 10

getIndex, 10

getLine, 8, 9

integrate

grid functions on overlapping grids, 36

method of analytic solutions, *see* twilight zone

OGFunction, 24

OGgetIndex, 10

OGPolyFunction, 28

OGPulseFunction, 31

OGTrigFunction, 29

reference counting

reference counted objects, 46

sPrintF, 7

sScanF, 7

surface integrals, 36

tridiagonal solver, 39

twilight zone

defining functions, 24

how to test PDE codes, 24

volume integrals, 36